Major Requirements of an OS

- Interleave the execution of several programs
  - to maximize utilization of CPU and other resources while providing reasonable response time
  - to support multiple user working interactively
  - for convenience (e.g., compile program while editing other file)
- Allocate resources required for execution of programs
- Support communication between executing programs

Previously, we listed several definitions of the term Process:

- A program in execution
- An instance of a program running on a computer
- A unit of execution characterised by
  - a single, sequential thread of execution
  - a current state
  - an associated set of system resources (memory, devices, files)
- Unit of resource ownership

Many applications consist of more than one thread of execution which share resources

⇒ distinction between thread and process

Processes and Threads

Process:

- “Owner” of resources allocated for individual program execution
- Can encompass more than one thread of execution
  - Outlook, Evolution: different threads for calendar, mail components etc

Thread:

- Unit of execution
- Belongs to a process
- Can be traced
  - list the sequence of instructions that execute

Example: Web Server
**EXAMPLE: Web Server**

- Dispatcher thread
- Worker thread
- Web page cache
- Network connection
- User space
- Kernel space

**Slide 5**

**Single-threaded Web Server Implementations**

- Sequential processing of requests:
  - web server gets request, processes it, accepts next request
  - CPU idle while data retrieved from disk
  - Poor performance
- Finite-State Machine:
  - use non-blocking read
  - program records state of current request
  - gets next event
  - on reply (signal) from disk, fetches and processes data
  - good performance, complicated to implement and debug

**Slide 6**

**Advantages of Threads**

1. Program does not stall when one of its operations blocks
   - save contents of a page to disk while downloading other page
2. Overhead for thread creation and destruction is less than for processes (depending on implementation, can be about a factor of 100 faster)
3. Simplification of programming model
4. Performance gains on machines with multiple CPU’s

**Slide 7**

**Threads and Processes**

- one process, one thread
- one process, multiple threads
- multiple processes, one thread per process
- multiple processes, multiple threads per process

**Slide 8**
THREADS AND PROCESSES

- Single process, single thread
  - MS-DOS, old MacOS
- Single process, multiple threads
  - OS/161 as distributed
- Multiple processes, single thread
  - traditional Unix
- Multiple processes, multiple threads
  - modern Unices (Solaris, Linux), Windows-2000

Note: Literature (incl. textbooks) often do not cleanly distinguish those concepts (for historical reasons!)

Logical traces of threads:

Thread A
- 5000: Starting address of code for Thread A
- 8000: Starting address of code for Thread B
- 12000: Starting address of code for Thread C

Thread B
- 5001
- 8001
- 12001

Thread C
- 5002
- 8002
- 12002
- 8003
- 12003
- 8004
- 12004

Thread States

Three states (may be more, depending on implementation):

1. Running: currently active, using CPU
2. Ready: runnable, waiting to be scheduled
3. Blocked: waiting for an event to occur (I/O, alarm)

Reasons for leaving the Running State

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
**Reasons for Leaving the Running State**

- Thread terminates
  - `exit()` system call (voluntary termination)
  - killed by another thread
  - killed by OS (due to exception)
- Thread cannot continue execution
  - blocked waiting for event (I/O)
- OS decides to give someone else a chance
  - requires the OS to be invoked
    - via system call or exception
    - via interrupt
- Thread voluntarily gives another thread a chance
  - `yield()` system call

**Non-running Threads**

- Many separate reasons for a thread not running
  - another thread is running on the CPU
  - thread is blocked (waiting for an event)
  - thread is in initialisation phase (during creation)
  - thread is being cleaned up (during `exit`, `kill`)
- Dispatching ought to be fast
  - Shouldn’t search through all threads to find runnable one
  - Achieved by distinguishing more thread states

**Separate Queues**

- Simplifies scheduler’s job
- How about `wakeup` of blocked thread when event occurs?

**Multiple wait queues:**

- Multiple events and queues for different events
Cooperative vs. Preemptive Multithreading

Cooperative multithreading:
- Threads determine exact order of execution
- Use `yield()` to switch between threads
- Problems if thread doesn’t yield (e.g., buggy)

Preemptive multitasking:
- OS preempts thread’s execution after some time
- Only guaranteed to work if H/W provides timer interrupt
- Implies unpredictable execution sequence
  - thread switch can happen between any two instructions
  - threads may require concurrency control

User-level Operations on Threads in OS/161

- Start a new thread in OS/161
  ```c
  thread_fork(const char * name,
  void * data1,
  unsigned long data2,
  void (* func)(void *, unsigned long),
  struct thread **ret);
  ```

- Terminate thread
  ```c
  thread_exit()
  ```

- Yield CPU
  ```c
  thread_yield()
  ```

Synchronisation:
- `thread_sleep(const void *addr)`
- `thread_wakeup(const void *addr)`